2014 IEEE Annual Symposium on Foundations of Computer Science

Outsourcing Private RAM Computation

Craig Gentry
IBM Research, T.J. Watson
cbgentry@us.ibm.com

Shai Halevi
IBM Research, T.J. Watson
shaih@alum.mit.edu

Mariana Raykova*
SRI International
mariana@cs.columbia.edu

Daniel Wichs**
Northeastern University
wichs@ccs.neu.edu

***Abstract*—We construct the first schemes that allow a client to privately outsource arbitrary program executions to a remote server while ensuring that: (I) the client's work is small and essentially independent of the complexity of the computation being outsourced, and (II) the server's work is only proportional to the run-time of the computation on a *random access machine (RAM)*, rather than its potentially much larger circuit size. Furthermore, our solutions are non-interactive and have the structure of *reusable garbled RAM programs*, addressing an open question of Lu and Ostrovsky (Eurocrypt 2013). We also construct schemes for an augmented variant of the above scenario, where the client can initially outsource a large private and persistent database to the server, and later outsource arbitrary program executions with read/write access to this database.**

Our solutions are built from *non-reusable garbled RAM* in conjunction with new types of *reusable garbled circuits* that are more efficient than prior solutions but only satisfy weaker security. For the basic setting without a persistent database, we can instantiate the required type of reusable garbled circuits from *indistinguishability obfuscation* or from *functional encryption for circuits* as a black-box. For the more complex setting with a persistent database, we can instantiate the required type of reusable garbled circuits using stronger notions of obfuscation. Our basic solution also requires the client to perform a one-time preprocessing step to garble a program at the cost of its RAM run-time, and we can avoid this cost using stronger notions of obfuscation. It remains an open problem to instantiate these new types of reusable garbled circuits under weaker assumptions, possibly avoiding obfuscation altogether.

We show several simple extensions of our results and techniques to achieve: efficiency proportional to the *input-specific* RAM run-time, *verifiability* of outsourced RAM computation, *functional encryption* for RAMs, and a candidate obfuscation for RAMs.

I. Introduction

Outsourcing computation from a weak client to a more powerful server is quickly becoming the predominant mode of day-to-day computation, bringing with it new security challenges and flourishing research into methods for addressing them. In this work we consider the challenge of *private outsourcing*, where the client wants to execute a program on a remote server while hiding from it the raw data to be used in the computation. Moreover, we want to ensure that:

1. The client should perform significantly less work than executing the program, and
2. The server should not have to do much more work than executing the program.

One method of outsourcing computation relies on fully homomorphic encryption (FHE) [RAD78], [Gen09], where the client simply encrypts her input and decrypts the output, and the server computes the program on encrypted data. Unfortunately, this solution requires the server to translate the program into a circuit and therefore work as hard as the *circuit size* of the computation, which in general, can be much larger than the work needed to execute the program on a random-access machine (RAM). In particular, even if we reach the zenith of FHE efficiency, with no overhead per homomorphic addition/multiplication, simply converting the computation into a circuit may already be too inefficient.

In general, a RAM computation with run-time T can have Turing-Machine run-time and circuit size as high as $\widetilde{O}(T^2)$, which is already a considerably large overhead [CR73], [PF79]. However, this distinction hardly tells the whole story, and the gap can be significantly larger in a setting involving program executions over a large persistent memory (e.g., a database). Consider for example the setting of private information retrieval (PIR) [CKGS98], where a server holds a large database of size N and a client wants to simply retrieve a single record from that database without the server learning the requested record. In this case, the RAM complexity of the retrieval query can be as low as $O(\log N)$, but the circuit complexity must be $\Omega(N)$ since the circuit must at least get the entire database as input, *a fully exponential gap*. The same gaps also already appear if we were to consider the Turing-Machine run-time of the computation instead of its circuit size.

We therefore would like to find an outsourcing protocol in which the server's work is only related to the RAM complexity of the program, while the client's work is essentially independent of the complexity of the program altogether. Furthermore, we would like to have such protocols in a setting where the client can initially outsource a large persistent memory (e.g., containing a database) and later outsource various RAM

*Research conducted in part while at IBM Research. Supported in part by NSF grants 1017660, 1421102

**Research conducted in part while visiting IBM Research. Supported in part by NSF grant 1347350

0272-5428/14 $31.00 © 2014 IEEE
DOI 10.1109/FOCS.2014.50

computations with read/write access to this memory.

Prior to this work, no such protocols were known. Although we do have private computation protocols over an outsourced memory based on *oblivious RAM* (ORAM) (e.g., [GO96], [OS97], [GKK+12]) where the server's work is proportional only to the RAM complexity of the computation, in all of these protocols the client also works as hard as the server. In particular, these protocols allow the client to save on storage by outsourcing the data to a remote server, but they do not provide *any* savings in computation over executing the program locally on local data.

In this work we describe *reusable garbled RAM* schemes, which offer the first solution to private outsourcing of RAM computation, where the server's work is only proportional to the RAM run-time of the computation and the client's work is essentially independent of the complexity of the computation altogether. In addition, these protocols are *non-interactive*, i.e., they only use one-way communication if the server is to learn the output (or two message communication if the client is to learn the output), making them useful even beyond outsourcing in "send and forget" settings.

A. Garbled Circuits and Garbled RAM

(Reusable) Garbled Circuits: Garbled circuits, introduced in the seminal work of Yao [Yao82], allow a client to garble a circuit C and then an input x in such a way that a server can use these garbled values to compute $C(x)$ without learning anything more about x. Until recently all the schemes that we had became insecure if the server ever got to see more than one garbled input per garbled circuit. Last year Goldwasser et al. described the first *reusable* circuit-garbling scheme [GKP+13b] where the client can garble a single circuit and then garble many inputs to that circuit without losing security. In this solution the client only needs to do a one-time pre-processing step to garble the circuit, at a cost proportional to the circuit size, but can then outsource many computations of this circuit on many different inputs by garbling them, with essentially no additional work per computation beyond what is needed to send the input.

Reusable Garbled TMs: The work of Goldwasser et al. [GKP+13a] extends the notion of reusable gabled circuits to Turing Machines (TMs). The main advantage is that the work of garbling a TM and the size of the garbled TM can be made proportional to the TM description size rather than the larger TM run-time or circuit size of the computation. In the context of outsourcing computation, this translates to getting rid of the pre-processing step so that the client *never* has to work as hard as evaluating the program. Another advantage of TMs over circuits is that TM computation can have much smaller "per-instance run-time" on some inputs than its "worst-case run-time". The solution of [GKP+13a] allows the server to run in time proportional to the per-instance TM run-time of the computation, at the security loss of leaking this run-time to the server. These advantages are mainly orthogonal to our main goal of allowing the server to work in time proportional to the RAM run-time of the computation, which could potentially be much smaller than the TM run-time or the circuit size. For example, when outsourcing a binary search over a large outsourced database of size N, both the circuit-size and the per-instance TM run-time will be $O(N)$ whereas the worst-case RAM run-time is $O(\log N)$.[1,2]

Garbled RAM. : Also recently, Lu and Ostrovsky introduced the notion of *garbled RAM* [LO13a]. Similar to garbled circuits, the client can garble a RAM program P, and later garble an input x in such a way that a server can use these garbled values to compute $P(x)$ without learning anything more about x. The complexity of garbling a RAM program (client complexity), the size of the garbled RAM, and the complexity of evaluating a garbled RAM (server complexity) are all proportional to the RAM run-time of the program rather than its circuit size.[3] The constructions of garbled RAM uses a clever combination of Yao garbled circuits and oblivious RAM (ORAM). Just like in Yao's circuits, the scheme is *not* reusable and becomes completely insecure if the server sees more than a single garbled input per garbled program. In other words, the client has to garble a fresh program for every computation, which requires as much work as doing the computation and therefore does not offer any savings in the context of outsourcing.

However, garbled RAM does offer an opportunity for amortization in the more complex setting involving multiple program executions over some persistent memory (e.g., a large outsourced database). The client can garble the initial memory contents once, and then can garble many different RAM programs and inputs (one input per program) that would be executed relative to the garbled memory, updating the memory with every execution. This property is called *persistent memory*, and allows the garbled memory to be *reused*. For example, the client can garble a large database of size N only once in time $O(N)$, and after that garble arbitrary queries to the database where the client work (time to garble a query) and the server work (time to evaluate a garbled

[1]The work of [GKP+13a] also constructs attribute-based encryption for RAM programs, but this scheme does not hide the input over which the RAM computation is performed. It cannot be used in the context of outsourcing private RAM computation.

[2]Although we mainly focus on our primary goal of having the server work in time proportional to the worst-case RAM run-time, we will also show a simple extension that reduces this to the per-instance RAM run-time along the lines of [GKP+13b], [GKP+13a].

[3]It was recently observed that the security proof for the scheme of [LO13a] has a subtle flaw, but the scheme can be fixed so as to get essentially the same properties as the original scheme [GHL+14].

query) are both proportional to the RAM run-time of the query. This provides a good solution for cases where the memory is large and the client wants to save on storage by outsourcing the contents, but the database queries are sufficiently simple that the client does not mind doing the work of the computation. It improves on simply using ORAM by making the program executions completely non-interactive. However, it still provides no savings in terms of client computation over having the client store the data and perform all computations locally.

Can Garbled RAM be Reusable? : The above raises the natural question whether we can obtain a *reusable garbled RAM*. In such a scheme, the client can garble a program once as a potentially expensive pre-processing step, and later outsource many arbitrary computations of this program to a server by efficiently garbling fresh inputs. The server can evaluate the garbled program on a garbled input in time proportional to the RAM complexity of the program. Furthermore, we would also like to do this in a setting where the client initially garbles a large *persistent memory* (e.g., database) and the programs can read/write to this memory. Such reusable garbled RAM schemes give a particularly nice solution to the problem of *outsourcing private RAM computation* with no interaction in the case where the server is to learn the output, and one round of back-and-forth communication when the client is to learn the output. The output can be made private from the server by simply garbling an augmented program that returns the output encrypted under the client's key. Furthermore, it can be made *verifiable* so that the client can be sure that the received output is correct, by simply garbling an augmented program that returns the output along with a message-authentication-tag of the output, under a key provided with the input.[4]

Outsourcing RAM vs. TM Computation: The work of Goldwasser et al. [GKP$^+$13a] shows how to outsource TM computation so that the server only works as hard as the per-instance TM run-time on the particular instance being outsourced, rather than the worst-case run-time. It is worth mentioning that the main goal of our work is orthogonal to the results in that paper, leverages the fact that TM computation can have much smaller "per-instance run-time" on some inputs than its "worst-case run-time". This advantage is lost when converting to a circuit. The solution of Goldwasser et al. [GKP$^+$13a] comes with an inherent security loss of leaking this per-instance run-time to the server. This is an orthogonal problem to our main result which considers worst-case run-time and does not allow the corresponding security loss (in which case the difference between TMs and circuits disappears). The challenges and techniques used to solve these two orthogonal problems also seem to be mostly unrelated. In the case of achieving per-instance TM run-time, the information revealed to the server is just the run-time of the computation, which has a small description, and therefore this does not capture the main challenge we face in the RAM setting. As one of the "extensions" of our main results, we also show how to achieve efficiency proportional to the per-instance run-time in the RAM model. This extension combines our main techniques with the techniques of Goldwasser et al. [GKP$^+$13a].

[4]We note that there are other approaches to verifiable RAM computation using SNARKs and proof-carrying data [Val08], [BCCT13], [BSCGT13], [BSCG$^+$13], [BFR$^+$13], but no other prior approaches that provide privacy. Therefore, we view the question of privacy as more pressing, but note that reusable garbled RAM gives us verifiability for free.

B. Our Solutions

In this work we describe the first solutions to the above problem of *reusable garbled-RAM*. We give three solutions with various tradeoffs between features/efficiency and the security assumptions needed to instantiate them.

As our **"basic"** solution, we describe a protocol that works in the setting *without* persistent memory, and requires the client to perform an expensive one-time pre-processing step to garble the program. In particular, the client can take a RAM program P with a run-time bound T and create a garbled version by working in time $\widetilde{O}(T)$. It can then very efficiently garble arbitrarily many inputs x_j to that program in time only proportional to the input (and output) size of the program, but independent of its complexity T. The server can evaluate the garbled program on each garbled input in time $\widetilde{O}(T)$. Furthermore, garbling new inputs only requires a *public key*, so anybody can outsource computations by creating garbled inputs to the garbled program.[5]

As our **"best-case"** solution, we describe a protocol that also works in the more complex setting involving persistent memory (e.g., database) and does not require any expensive pre-processing. Specifically, in this solution the client has the option to garble some persistent memory of size N in time $\widetilde{O}(N)$. It can then garble a RAM program P in time proportional to its description length $O(|P|)$ but *independent of its running time*. Finally it can garble many "short inputs" x_i to the program P in time proportional to the input (and output) size of the program. The server can evaluate the garbled

[5]By default, we do not require "program privacy" and allow the server to learn the description of the outsourced program. In the public-key setting this is inherent since the server can create garbled inputs on his own and therefore learn information about the functionality of the program. We only guarantee that the server does not learn anything about the inputs that are garbled by the client, beyond the output of the computation. We will describe a simple extension that achieves program privacy, but necessarily moves the construction to the secret key setting where only the client that creates the garbled program can create garbled inputs.

program with the garbled input over the garbled memory in time proportional to the program's RAM run-time $\widetilde{O}(T)$. For example, the programs P could be a SQL database implementation and the inputs x_i could specify various complex database queries. We stress that the program executions can both read and write to memory, and that the changes to memory made by one program execution persist for the next program execution and cannot be "rolled back" by a malicious server.

Assumptions. : Our main contribution is to reduce the complex problems of reusable garbled *RAM* to seemingly simpler problems dealing with reusable garbled *circuits*, and avoiding the complexity of RAM altogether. Each of the above three constructions corresponds to a new notion of security/efficiency for reusable garbled circuits, which may be of independent interest (see below). Ultimately, we can instantiate these new notions of reusable garbled circuits using various obfuscation-based assumptions. The garbled circuits that are used in our "basic" solution can be based on indistinguishability obfuscation or on the existence of indistinguishability-secure functional encryption for circuits. In particular, the latter is a falsifiable assumption. The reusable garbled circuits needed for our "middle" and "best-case" solutions can be based on stronger variants of obfuscation, related to differing-inputs obfuscation. We stress that the use of obfuscation does not seem inherent, and there is hope that these new notions of reusable garbled circuits could be instantiated under simpler assumptions that avoid obfuscation altogether. Indeed, this seems to be an easier problem than the related problem of achieving indistinguishability-based security for functional encryption without obfuscation.

C. Our Techniques

We obtain reusable garbled RAM from a combination of non-reusable garbled RAM, and a new form of reusable garbled circuits whose properties we discuss shortly.

Our solutions are based on a very simple intuitive idea: given a RAM program P, consider the circuit $C[P]$ which has P hard-coded in its description, gets as input (r, x), and uses r as randomness to created a one-time garbled program $\tilde{P}_{one}$ (garbling P) and a garbled input $\tilde{x}_{one}$ (garbling x). Garbled RAM ensures that the circuit-size of $C[P]$ is only dependent on the RAM run-time T of the program P rather than its potentially much larger circuit size. Our main idea is to create a reusable garbled circuit $\tilde{C}_{reuse}$ of the circuit $C[P]$, which the client gives to the server. Each time the client wants to run a new program execution with input x_i, she chooses some fresh randomness r_i, and garbles (r_i, x_i) under the reusable circuit garbling scheme. The server runs the reusable garbled circuit $\tilde{C}_{reuse}$ on the garbled input from the client to *create* a one-time garbled RAM program $\tilde{P}_{one}$ and garbled input $\tilde{x}_{one}$, and then *evaluates* $\tilde{P}_{one}$ on $\tilde{x}_{one}$.

The above idea is not entirely new, but it turns out that it cannot quite work right out of the box.[6] Notice that the circuit $C[P]$ above has a huge output of size $\widetilde{O}(T)$ even though its input is small. Unfortunately, the reusable garbled circuit construction of Goldwasser et al. [GKP$^+$13b], requires that the size of the *garbled input* to the circuit always exceeds the size of the circuit's *output*, even if the size of the *actual input* of the circuit is small.[7] We call this property *output-size dependence*. In particular, to securely garble a short input (r_i, x_i), the client would have to create a huge garbled input of size $\widetilde{O}(T)$, which would require that the client works at least as hard as evaluating the program, and completely obliterate the efficiency benefits of outsourcing. Unfortunately, this is also not an accidental property of the construction of [GKP$^+$13b] and we show that *any* reusable circuit garbling scheme with simulation-based security must have output-size dependence (see the full version [GHRW14]).

Our main observation is that we do not necessarily require full simulation-based security from the reusable garbled circuit component, even though we insist on achieving full simulation-based security for the final reusable garbled RAM construction. We come up with new notions of security for reusable garbled circuits that we call "distributional indistinguishability" (with two flavors), which turn out to suffice in our constructions and may be of independent interest elsewhere. Intuitively, these notions say that one cannot distinguish garbled inputs from two distributions that produce indistinguishable outputs. Moreover, these weaker security notions seem to plausibly allow for more efficient constructions that avoid "output-size dependence". Indeed, we propose new candidate constructions of such reusable garbled circuits based on obfuscation. Our two constructions of reusable garbled RAM translate to two flavors of reusable garbled circuits with "distributional indistinguishability". The weaker flavor can be based on indistinguishability obfuscation while the stronger one seems to require stronger obfuscation-based assumptions. It remains as an open problem to achieve these notions from other assumptions, ideally avoiding obfuscation altogether.

D. Extensions

In the full version [GHRW14] we explore several extensions and applications of our main results and

[6] A variant of an idea along these lines appeared in an early version of [LO13a] and was outlined in a rump-session talk [LO13b] but was retracted for exactly the reasons we describe here.

[7] The scheme and parameters of [GKP$^+$13b] are described for circuits with 1-bit output, but can easily be extended to the setting of multi-bit output at the above cost of having the size of the garbled-input grow with the output size.

techniques. We discuss how to generically augment reusable garbled RAM to get *output privacy* (server does not learn the output of the computation) and *verifiability* (client can be certain that the received output is correct). We also discuss how to get *program privacy* where the server does not learn the code of the program. Furthermore, we discuss how to leverage our solutions to get *input-specific run-time* where the server's work is only proportional to the RAM run-time of $P(x)$ on the desired input x rather than the worst-case run-time of P on inputs of size n. We also discuss applications to MPC where only one party needs to work as hard as the program's RAM run-time. In the full version [GHRW14], we also show how to leverage our techniques to build indistinguishability-secure functional encryption (FE) for RAM programs (without persistent data) using FE for circuits as a black box. We show that this gives an alternate construction of reusable garbled RAM from FE for circuits, without using obfuscation directly. However, the only known construction of FE for circuits in [GGH$^+$13] relies on iO. Lastly, we propose a speculative candidate construction for obfuscating RAMs using obfuscation for circuits and conjecture that it can achieve iO for RAMs.

II. Preliminaries

The two models of computation that we deal with in this work are circuits and RAM programs. Intuitively, a RAM program has access to some memory of size N and each step of the program can read/write to an arbitrary location of memory. We usually assume that the memory starts out empty. However, when we consider program executions over a persistent memory/database, it is useful to consider the case where the memory initially contains some data D. We use the notation $P^D(x)$ to denote the execution of a program P with random-access memory containing D and a short input x. For the RAM programs we consider in this work, we assume that we have an absolute bound on their worst-case running time, input/output length, and memory usage. A somewhat more detailed specification of the RAM model is found in the full version [GHRW14].

We use $C[\mathsf{prm}]$ or $P[\mathsf{prm}]$ to denote a circuit/program that depends on a parameter prm. The parameter can be an arbitrary string, and can itself be another circuit or program. We think of prm as being "hard wired" in the description of the corresponding circuit/program. The input to a circuit/program is specified inside parenthesis, so $C[\mathsf{prm}](x)$ describes the computation of the circuit $C[\mathsf{prm}]$ (whose definition depends on prm) on the input x.

III. Reusable GRAM without Persistent Memory

A. Definitions

We begin by defining non-reusable (one-time) and reusable garbled RAM. The syntax of the scheme is the same in both cases, and the difference is only in the security requirements.

Definition III.1 (GRAM without persistent memory)**.** *A* garbled RAM *scheme without persistent memory consists of procedures* $\mathsf{GR} = (\mathsf{GR.prog}, \mathsf{GR.inp}, \mathsf{GR.eval})$*:*

- $(\tilde{P}, s) \leftarrow \mathsf{GR.prog}(1^\lambda, P, (n, m, t))$ *: Gets a RAM program P, and bounds on the program's: input size n, output size m, and run-time t (say that all bounds encoded in binary). Outputs a garbled program $\tilde{P}$ and a garbling key s.*
- $\tilde{x} \leftarrow \mathsf{GR.inp}(x, s, (n, m, t))$ *: Takes as input an n-bit value x, the garbling key s the same bounds (n, m, t). It outputs the garbled input $\tilde{x}$.*
- $y = \mathsf{GR.eval}(\tilde{P}, \tilde{x})$*: This is a RAM program that takes $(\tilde{P}, \tilde{x})$ as input and computes the output y.*

We require that for any program P with parameters (n, m, t), any input $x \in \{0,1\}^n$ if $\tilde{P}, \tilde{x}$ are created as described, then $\mathsf{GR.eval}(\tilde{P}, \tilde{x}) = P(x)$ *with probability 1.*

Definition III.2 (GRAM Security)**.** *Let* GR *be a garbled RAM scheme as above.*

- GR *has reusable security if there exists a PPT simulator* Sim *such that, for all RAM programs P with polynomial parameters (n, m, t) and all polynomial-length input-vectors $(x_1, \ldots, x_q)$, the following distributions are computationally indistinguishable:*

  $$(\tilde{P}, \tilde{x}_1, \ldots, \tilde{x}_q) \overset{\text{comp}}{\approx} \mathsf{Sim}(1^\lambda, P, (n, m, t), y_1, \ldots, y_q)$$

  where $(\tilde{P}, s) \leftarrow \mathsf{GR.prog}(1^\lambda, P, (n, m, t)), \tilde{x}_i \leftarrow \mathsf{GR.inp}(x_i, s, (n, m, t))$ *and* $y_i = P(x_i)$*. The simulator is required to run in time* $\mathsf{poly}(\lambda, |P|, n, m, t, q)$*.*
- GR *has* security with public input garbling *if there exists a PPT simulator* Sim' *such that the above security holds even when including the input-garbling key s in the left-hand distribution,*

  $$(\tilde{P}, s, \tilde{x}_1, \ldots, \tilde{x}_q) \overset{\text{comp}}{\approx} \mathsf{Sim}'(1^\lambda, P, (n, m, t), y_1, \ldots, y_q).$$

- GR *has* one-time *security (resp. one-time security with public input garbling) if the above only holds for $q = 1$.*

Efficiency: We require that: GR.prog runs in time $\widetilde{O}(|P| + n + m + t) \cdot \mathsf{poly}(\lambda)$ and can be though of as a one-time preprocessing where the client has to work as hard as the program execution. GR.inp runs in time $\widetilde{O}(m+n) \cdot \mathsf{poly}(\lambda)$, and therefore is asymptotically

efficient even as t becomes large. $\mathsf{GR.eval}$ runs in time $\widetilde{O}(|P|+n+m+t)\cdot \mathsf{poly}(\lambda)$ on a RAM, and therefore is only linear in the original program's running time t. In addition, we require that $\mathsf{GR.prog}, \mathsf{GR.inp}$ can be expressed as *circuits* with the above bounds denoting their circuit size. On the other hand, $\mathsf{GR.eval}$ is crucially expressed as a RAM program.

Remark on Program Privacy: Note that our definition does not explicitly consider *program privacy* and we assume that the code of the program P is public. This can be fixed via standard transformations (see the full version).

We rely on prior constructions of one-time garbled RAM schemes [LO13a], [GHL$^+$14].[8]

Theorem III.3 ([LO13a], [GHL$^+$14])**.** *Assuming the existence of selectively-secure identity-based encryption (IBE), there exist garbled RAM schemes with* one-time *security satisfying the above efficiency requirements.*

Garbled Circuits: As a useful tool, we will rely on the notion of reusable garbled circuits and we define the syntax of such schemes as follows.

Definition III.4 (Garbled Circuits)**.** A *garbled circuit* scheme consists of three procedures, $\mathsf{GC} = (\mathsf{GC.circ}, \mathsf{GC.inp}, \mathsf{GC.eval})$:

- $(\tilde{C}, s) \leftarrow \mathsf{GC.circ}(1^\lambda, C)$: Gets an input circuit C and outputs a garbled circuit $\tilde{C}$ and key s.
- $\tilde{x} \leftarrow \mathsf{GC.inp}(x, s)$: Gets an input x and the same key s. Outputs the garbled input $\tilde{x}$.
- $y \leftarrow \mathsf{GC.eval}(\tilde{C}, \tilde{x})$: Gets a garbled circuit $\tilde{C}$ and matching input $\tilde{x}$, and computes the output.

We require that for any circuit C, input x, setting $(\tilde{C}, s) \leftarrow \mathsf{GC.circ}(1^\lambda, C)$ and $\tilde{x} \leftarrow \mathsf{GC.inp}(x, s)$ we get $\mathsf{GC.eval}(\tilde{C}, \tilde{x}) = C(x)$.

We defer discussion of the security properties that we need until later.

Output-size independent efficiency: Our main efficiency requirement is that $\mathsf{GC.inp}$ works in time $|x| \cdot \mathsf{poly}(\lambda)$, in particular its running time can *only* depend on the input size and not on the circuit size or even the output size. We call this requirement *output-size independent efficiency*. In addition, we require that $\mathsf{GC.circ}$ works in time $|C| \cdot \mathsf{poly}(\lambda)$ and that $\mathsf{GC.eval}$ works in time $\widetilde{O}(|\tilde{C}| + |\tilde{x}|) \cdot \mathsf{poly}(\lambda) = \widetilde{O}(|C| + |x|) \cdot \mathsf{poly}(\lambda)$.

B. Construction of Reusable GRAM

Overview: We construct a reusable garbled-RAM scheme by combining reusable garbled circuit with a one-time garbled RAM scheme. Let $\mathsf{GC} = (\mathsf{GC.circ}, \mathsf{GC.inp}, \mathsf{GC.eval})$ be a reusable garbled circuit scheme whose required security properties we specify later and $\mathsf{GR1} = (\mathsf{GR1.prog}, \mathsf{GR1.inp}, \mathsf{GR1.eval})$ be a one-time garbled-RAM scheme. Recall our first approach from the introduction, which was to consider the circuit $C[P](r, x)$ that has P hard-wired in and gets as input randomness r and input x. The circuit $C[P](r, x)$ runs $(\tilde{P}_{one}, s) \leftarrow \mathsf{GR1.prog}(1^\lambda, P, (\cdots))$ and $\tilde{x}_{one} \leftarrow \mathsf{GR1.inp}(x, s, (\cdots))$, using r as randomness, and outputs $(\tilde{P}_{one}, \tilde{x}_{one})$. Our hope was to create a reusable garbled circuit $\tilde{C} \leftarrow \mathsf{GC.circ}(C[P])$ as our reusable garbled RAM program.

Observe that the circuit $C[P]$ from above has short input and very long output, related to the running time of P. Unfortunately, the construction of reusable garbled circuits of Goldwasser et al. [GKP$^+$13b], requires that the size of the *garbled input* to the circuit always exceeds the size of the circuit's *output*, even if the size of the *actual input* of the circuit is small. In particular, they have output-size dependence. In the full version, we show that *any* reusable circuit garbling scheme with simulation-based security must have output-size dependence. In our context, that would mean that garbling an input to the program takes as much time as evaluating a program and therefore would not be useful in the context of outsourcing.

We fix the problem above by tweaking the above construction in a way that allows us to reduce the security requirement on the reusable garbled circuit to something weaker than simulation security, while still achieving simulation security for our final construction. In particular, we first present our modified construction of reusable garbled RAM from reusable garbled circuits, then we present a new notion of security for reusable garbled circuits that we call "distributional indistinguishability", and show that is suffices to make our construction secure (Section III-C) and finally, we show how to instantiate this new notion of reusable garbled circuits (Section III-D) while achieving output-size independent efficiency.

The main modification that we make to the first-attempt construction from above is to first transform a program P with run-time t into a modified program P^+ that we call the *real-or-dummy* program. In addition to the input x, the new program P^+ takes also an alleged output y and a flag ψ. If $\psi = 1$ (real) then $P^+(x, \psi, y)$ simply executes $P(x)$, ignoring y. If If $\psi = 0$ (dummy), on the other hand, then P^+ simply executes t dummy steps and outputs y, ignoring x.

Just as before, we consider the circuit $C[P^+](r, (x, \psi, y))$ that outputs a one-time garbled program $\tilde{P}$ garbling P^+ and garbled input $\tilde{x}$ garbling (x, ψ, y) using r as randomness. We then construct a reusable garbled circuit $\tilde{C}$ garbling $C[P^+]$ as the reusable garbled RAM program. Intuitively, the simulator of the reusable garbled RAM will

[8]The syntax of [GHL$^+$14] includes a separate procedure $\mathsf{GR.data}$ to garble memory, but for now we can think of this as part of the $\mathsf{GR.prog}$ procedure. We mention that similar result with slightly worse efficiency can be achieved under one-way functions.

simply provide a garbling of a "dummy input" consisting of $(r, 0^n, \psi = 0, y)$ instead of the "real" input $(r, x, \psi = 1, 0^m)$. Proving security of the new construction boils down to proving that, given $\tilde{C}$, one cannot distinguish many garbling of real inputs vs. dummy inputs. Notice that the outputs $\tilde{P}, \tilde{x}$ derived from real-inputs vs. dummy-inputs look indistinguishable by the security of the one-time garbled RAM. Therefore, we reduce simulation-based security of the full scheme to showing a new type of "distributional indistinguishability" for reusable garbled circuits, where it should be hard to distinguish garbled inputs from two different distributions (e.g., read or dummy) that produce indistinguishable outputs. This idea is similar in spirit to one used by De Caro et al. [CIJ+13] to convert indistinguishability-based security to simulation-based security for functional encryption. However, our notion of "distributional indistinguishability" is new.

The real-or-dummy program $\mathbf{P}^+$: In more detail, for a RAM program P with input-size n, output-size m and running-time bound t, let P^+ be a RAM program that gets as input (x, ψ, y) with $|x| = n$, $|\psi| = 1$ and $|y| = m$. If $\psi = 1$ then $P^+(x, \psi, y)$ outputs $P(x)$, and if $\psi = 0$ then it outputs y after t steps. Note that the complexity of P^+ is essentially the same as P, except that it has input of size $n + m + 1$ rather than just n.

The program-garbling circuit: A central component of our construction is a circuit that runs the program- and input-garbling routines of the underlying one-time GRAM scheme. For a RAM program P with input-size n, output-size m and running-time bound t, and for security parameter λ, define $C[P, n, m, t, \lambda]$ as the following circuit with $n + m + 2\lambda + 1$ input bits:[9]

$\underline{\mathbf{C}[\mathbf{P}, \mathbf{n}, \mathbf{m}, \mathbf{t}, \lambda](r, x, \psi, y)}$:
// $r = (r_1, r_2) \in \{0,1\}^{2\lambda}$,
// $x \in \{0,1\}^n$, $\psi \in \{0,1\}$, $y \in \{0,1\}^m$
1. Run $(\tilde{P}, s) \leftarrow \mathsf{GR1.prog}(1^\lambda, P^+, (n, m, t); r_1)$,
$\tilde{x} \leftarrow \mathsf{GR1.inp}((x, \psi, y), s, (n, m, t); r_2)$,
2. Output $(\tilde{P}, \tilde{x})$.

Recall that for the program P^+ as above, the circuit-size of GR1.prog is $\widetilde{O}(|P| + n + m + t) \cdot \mathrm{poly}(\lambda)$ and the circuit-size of GR1.inp is $O(n+m) \cdot \mathrm{poly}(\lambda)$. Hence the size of $C[P, n, m, t, \lambda]$ (as well as the time that it takes to generate its description) can be bounded by $\widetilde{O}(|P| + n + m + t) \cdot \mathrm{poly}(\lambda)$.

GR*: Reusable Garbled RAM Construction:* We describe our reusable GRAM construction $\mathsf{GR} = (\mathsf{GR.prog}, \mathsf{GR.inp}, \mathsf{GR.eval})$ in the following figure.

[9]For simplicity, we assume that GR1.prog, GR1.inp uses exactly λ bits of randomness each. This can always be made the case by using a pseudorandom generator.

$\underline{\mathsf{GR.prog}(1^\lambda, P, (n, m, t))}$:
1. Construct the circuit $C[P, n, m, t, \lambda]$ shown above.
2. Output a garbling of this circuit
$(\tilde{C}, s) \leftarrow \mathsf{GC.circ}(1^\lambda, C[P, n, m, t, \lambda])$.
$\underline{\mathsf{GR.inp}(x, s, (n, m, t))}$:
1. Choose a random $r \leftarrow \{0,1\}^{2\lambda}$,
set $\psi = 1, y = 0^m$, and $w = (r, x, \psi, y)$,
2. Garble the input to $C[P \ldots]$, outputting
$\tilde{w} \leftarrow \mathsf{GC.inp}(w, s)$.
$\underline{\mathsf{GR.eval}(\tilde{C}, \tilde{w})}$:
1. Evaluate $(\tilde{P}, \tilde{x}) \leftarrow \mathsf{GC.eval}(\tilde{C}, \tilde{w})$,
// $= C[P, n, m, t, \lambda](s, x, \psi, y)$
2. Evaluate the 1-time GRAM and output
$y \leftarrow \mathsf{GR1.eval}(\tilde{P}, \tilde{x})$. // $= P(x)$

Functionality and complexity: The correctness of this scheme can be verified by inspection. As for its complexity, since the size of $C[P, n, m, t, \lambda]$ and the time to construct it are bounded by $\widetilde{O}(|P| + n + m + t) \cdot \mathrm{poly}(\lambda)$, the overall complexity of GR.prog is also $\widetilde{O}(|P|+n+m+t) \cdot \mathrm{poly}(\lambda)$. The input to $C[P, n, m, t, \lambda]$ has length $O(n + m + \lambda)$, and therefore the time to garble that input (which is the complexity of GR.inp) is bounded by $O(n + m) \cdot \mathrm{poly}(\lambda)$. [10] Finally, the time to evaluate the garbled circuit is polynomial in its size, hence the first step of GR.eval has complexity $\widetilde{O}(|P| + n+m+t) \cdot \mathrm{poly}(\lambda)$. Also, the time to evaluate a garbling of P^+ is essentially the running time $\mathsf{GR1.eval}(\tilde{P}, \tilde{x})$ is essentially that of P^+, so also the second step has complexity bounded by $\widetilde{O}(|P| + n + m + t) \cdot \mathrm{poly}(\lambda)$ as well, and so this term bounds the overall complexity of GR.eval.

C. Simulation Security From Distributional Indistinguishability

A crucial observation is that we can prove simulation-security for the above reusable GRAM construction GR using a new notion of "distributional indistinguishability" security for the underlying garbled circuit scheme and the usual simulation security for the underlying one-time GRAM scheme. "Distributional indistinguishability' says that one cannot distinguish garbled inputs from any two sets of independent distributions that produce individually indistinguishable outputs.

Definition III.5. *Let* $\mathsf{GC} = (\mathsf{GC.circ}, \mathsf{GC.inp}, \mathsf{GC.eval})$ *be a garbled circuit scheme. We say that* GC *provides* distributional indistinguishability *if for every circuit ensemble $C = \{C_\lambda\}$, every polynomial $q = q(\lambda)$, and every $2q$ polynomial-time samplable distributions $D_1, \ldots, D_q$ and $D'_1, \ldots, D'_q$, if for all $j \in [q]$ it holds*

[10]Note that the complexity of input garbling depends on *both the input and the output size of P*, meaning that our overall construction has output-size dependence. This is necessary for reusable simulation-security (see the full version [GHRW14]).

that $C(w_j) \stackrel{\text{comp}}{\approx} C(w'_j)$ *where* $w_j \leftarrow D_j(1^\lambda), w'_j \leftarrow D'_j(1^\lambda)$ *then it also holds that*

$$\langle \tilde{C}, \tilde{w}_1, \ldots \tilde{w}_q \rangle \stackrel{\text{comp}}{\approx} \langle \tilde{C}, \tilde{w}'_1, \ldots \tilde{w}'_q \rangle$$

where $(\tilde{C}, s) \leftarrow \mathsf{GC.circ}(1^\lambda, C_\lambda)$, $w_i \leftarrow D_i(1^\lambda)$, $w'_i \leftarrow D'_i(1^\lambda)$, $\tilde{w}_i \leftarrow \mathsf{GC.inp}(w_i, s)$, $\tilde{w}'_i \leftarrow \mathsf{GC.inp}(w'_i, s)$.

We say that the scheme has security with public input garbling *if the above holds when we include the garbling key* s *in the two distributions on the bottom.*

We remark that this notion is clearly implied by simulation security for reusable garbled circuits, but simulation security of reusable garbled circuits requires "output-size dependence" where the size of the garbled input must exceed that of the circuit's output (see the full version [GHRW14]). Furthermore, we remark that for any scheme with public input garbling, distributional indistinguishability for $q = 1$ implies security for arbitrary q by a simple hybrid argument. Interestingly, this does not hold for simulation security where it may be possible to simulate $q = 1$ inputs but not possible to simulate for a larger q, even if the scheme has public input garbling.

In Section III-D we show how to construct a reusable circuit-garbling scheme with output-size independence satisfying this definition using indistinguishability obfuscation.

Theorem III.6. *If* $\mathsf{GC} = (\mathsf{GC.circ}, \mathsf{GC.inp}, \mathsf{GC.eval})$ *is a reusable garbled-circuit scheme satisfying distributional indistinguishability and output-independent efficiency, and* $\mathsf{GR1} = (\mathsf{GR1.prog}, \mathsf{GR1.inp}, \mathsf{GR1.eval})$ *is a one-time garbled-RAM scheme, then the scheme* GR *from above is a reusable garbled-RAM scheme satisfying simulation security. Furthermore, if* GC *has security with public input garbling, then so does* GR.

Proof: The simulator was sketched above: On input $(1^\lambda, (n, m, t), P, y_1, \ldots, y_q)$ with $|y_i| = m$ for all i, the simulator GR.sim begins just as the garbling procedure of the actual scheme, namely by constructing the circuit $C[P, n, m, t, \lambda]$ and applying to it the circuit-garbling procedure to get $(\tilde{C}, s) \leftarrow \mathsf{GC.circ}(1^\lambda, C[P, n, m, t, \lambda])$. Next, for every y_i the simulator chooses a uniformly random $r_i \in \{0,1\}^{2\lambda}$, sets $w'_i = (r_i, 0, \psi = 0, y_i)$ and $\tilde{w}'_i \leftarrow \mathsf{GC.inp}(w_i, s)$. The output of the simulator GR.sim consists of $(\tilde{C}, \tilde{w}'_1, \ldots, \tilde{w}'_q)$.

We next prove indistinguishability between the real and simulated outputs. Fix the program P and inputs $x_1, \ldots, x_q$ for P, and denote $y_i = P(x_i)$ for all i. Let $w_i = (r_i, x_i, \psi = 1, 0^m)$ where $r_i \in \{0,1\}^{2\lambda}$. We first argue that the output distributions on inputs w_i and w'_i are indistinguishable.

Claim III.7. *Denote* $C = C[P, n, m, t, \lambda]$. *If the scheme* $\mathsf{GR1} = (\mathsf{GR1.prog}, \mathsf{GR1.inp}, \mathsf{GR1.eval})$ *satisfies one-time GRAM security then for every* $x \in \{0,1\}^n$ *and* $y = P(x)$ *we have*

$$\{C(w_i)\} \stackrel{\text{comp}}{\approx} \{C(w'_i)\},$$

where w_i, w'_i *are chosen as described above.*

Proof: Note that $C(w_i) = (\tilde{P}_i, \tilde{x}_i)$ and $C(w'_i) = (\tilde{P}_i, \tilde{x}'_i)$ where $\tilde{P}_i$ is a garbled version of the program P^+, $\tilde{x}_i$ is a garble version of the input $(x_i, 1, 0^m)$ and $\tilde{x}'_i$ is a garbled version of the input $(0^n, 0, y_i)$. The one-time simulation-security of the underlying GR1 scheme implies that:

$$(\tilde{P}_i, \tilde{x}_i) \stackrel{\text{comp}}{\approx} \mathsf{GR1.Sim}(1^\lambda, P, (n, m, t), y_i) \stackrel{\text{comp}}{\approx} (\tilde{P}_i, \tilde{x}'_i)$$

since $y_i = P^+((x_i, 1, 0^m)) = P^+((0^n, 0, y_i))$. This proves the claim. ∎

Claim III.7 implies that the distributions of the w_i, w'_i's satisfy the condition of Definition III.5, and by the distributional indistinguishability of GC we conclude that also

$$\langle \tilde{C}, \tilde{w}_1, \ldots \tilde{w}_q \rangle \stackrel{\text{comp}}{\approx} \langle \tilde{C}, \tilde{w}'_1, \ldots \tilde{w}'_q \rangle.$$

This completes the proof, since these are exactly the output distributions of the scheme GR and its simulator GR.sim. ∎

D. Achieving Distributional Indistinguishability

We now construct reusable garbled circuits with "distributional indistinguishability" and "output-size independent efficiency". Furthermore, our construction has public input-garbling. The construction is based on "indistinguishability obfuscation" (see the full version [GHRW14]) and a NIZK which is "statistically simulation sound" (see the full version [GHRW14]). It is inspired by the construction of functional encryption from indistinguishability obfuscation of [GGH+13]. For efficiency, we will require that the obfuscation of a circuit C has size $|C| \cdot \mathsf{poly}(\lambda)$ linear in the size of the original circuit, and this is indeed the case for candidate schemes (see the full version [GHRW14] for details). However, we also note that this requirement is not crucial for our full construction and we can get qualitatively similar final results using iO with any polynomial blow-up in circuit size (see the full version [GHRW14] for details). Nevertheless, assuming obfuscation with linear slow-down makes things simpler and therefore we use this as our default notion.

Construction: Let $\mathcal{O}$ be an obfuscation scheme, let $\mathcal{PKE} = (\mathsf{Setup}, \mathsf{Encrypt}, \mathsf{Decrypt})$ be a public key encryption scheme, and let $\Pi = (K, P, V)$ be a NIZK scheme with statistical simulation soundness. Let L_{EQ} be the NP language defined as

$$L_{EQ} = \{ \ (\mathsf{pk}_1, \mathsf{pk}_2, c_1, c_2) \ : \ \exists \ m, r_1, r_2, \\ c_1 = \mathsf{Encrypt}(\mathsf{pk}_1, m; r_1) \wedge c_2 = \mathsf{Encrypt}(\mathsf{pk}_2, m; r_2)\}.$$

For any circuit $C : \{0,1\}^n \to \{0,1\}^m$ define the circuit $C^*[\sigma, \mathsf{pk}_1, \mathsf{pk}_2, b, \mathsf{sk}, u, v]$ where: σ is a CRS for the NIZK, $\mathsf{pk}_1, \mathsf{pk}_2$ are encryption keys, $b \in \{1,2\}$ is an index, sk is the decryption key for pk_b, u is of size $|(c_1, c_2, \pi)|$ where c_1, c_2 are ciphertexts of n-bit messages and π is a NIZK for L_{EQ}, and $v \in \{0,1\}^m$.

$C^*[\sigma, \mathsf{pk}_1, \mathsf{pk}_2, b, \mathsf{sk}, u, v](c_1, c_2, \pi)$:
1. If $u = (c_1, c_2, \pi)$ output v.
2. Verify that π is a proof of $(\mathsf{pk}_1, \mathsf{pk}_2, c_1, c_2) \in L_{EQ}$ by running $V(\sigma, (\mathsf{pk}_1, \mathsf{pk}_2, c_1, c_2), \pi)$.
If this rejects, output $\perp$.
3. Compute $x = \mathsf{Decrypt}(\mathsf{sk}, c_b)$. Output $C(x)$.

We define the circuit garbling scheme $\mathsf{GC} = (\mathsf{GC.circ}, \mathsf{GC.inp}, \mathsf{GC.eval})$, which has public input garbling, as follows:

- $(\tilde{C}, s) \leftarrow \mathsf{GC.circ}(1^\lambda, C)$: Generate $(\mathsf{pk}_1, \mathsf{sk}_1) \leftarrow \mathsf{Setup}(1^\lambda)$, $(\mathsf{pk}_2, \mathsf{sk}_2) \leftarrow \mathsf{Setup}(1^\lambda)$, $\sigma \leftarrow K(1^\lambda)$. Construct the circuit $C^* := C^*[\sigma, \mathsf{pk}_1, \mathsf{pk}_2, 1, \mathsf{sk}_1, u = \perp, v = \perp]$ from C as shown. Output $\tilde{C} \leftarrow \mathcal{O}(C^*)$ and $s := (\sigma, \mathsf{pk}_1, \mathsf{pk}_2)$.
- $\tilde{x} \leftarrow \mathsf{GC.inp}(x, s)$: output $\tilde{x} := (c_1, c_2, \pi)$, where $c_1 \leftarrow \mathsf{Encrypt}(\mathsf{pk}_1, x; r_1)$, $c_2 \leftarrow \mathsf{Encrypt}(\mathsf{pk}_2, x; r_2)$ and $\pi \leftarrow P(\sigma, (\mathsf{pk}_1, \mathsf{pk}_2, c_1, c_2), (r_1, r_2))$ is an NIZK that $(\mathsf{pk}_1, \mathsf{pk}_2, c_1, c_2) \in L_{EQ}$.
- $\mathsf{GC.eval}(\tilde{C}, \tilde{x})$: Interpret $\tilde{C}$ as an obfuscated circuit and output $\tilde{C}(\tilde{x})$.

Theorem III.8. *If $\mathcal{O}$ is an indistinguishability obfuscator, Π is a statistical-simulation-sound (SSS) NIZK, and $\mathcal{PKE}$ is a semantically secure encryption scheme, then the above construction GC is a* reusable garbled circuit with distributional indistinguishability *and* public input garbling.

A proof for Theorem III.8, using "punctured programs" paradigm, is given in the full version of the paper [GHRW14].

Summary: Combining the above with Theorem III.3 and III.6, and the facts that indistinguishability obfuscation + one-way function implies selectively secure functional encryption which implies IBE [GGH+13], and that statistically simulation sound NIZK can be constructed from statistically sounds NIZK [GGH+13], we get the following corollary.

Corollary III.9. *If indistinguishability obfuscation, one-way functions, and statistically sounds NIZKs exist, then there exists a reusable garbled-RAM scheme without persistent memory satisfying Definition III.2. Furthermore, it supports public input garbling.*

IV. Garbled RAM with Persistent Memory

Due to space constraints we defer the definition and construction of reusable garbled RAM with persistent memory to the full version of the paper [GHRW14].

V. Conclusions

We have shown how to privately outsource RAM computation from a weak client to a more powerful server via reusable garbled RAM schemes. Our main contribution was to reduce the problem of reusable garbled RAM into seemingly simpler problems dealing with reusable garbled circuits. In doing so, we introduced new notions of security for such garbled circuit that we call "distributional indistinguishability" and "correlated distributional indistinguishability" which may be of independent interest and seem to allow for greater (output-size independent) efficiency than the stronger simulation-based security. Lastly, we showed how to construct such schemes under obfuscation-based assumptions. The main open problem is to provide constructions of such reusable garbled circuits under weaker assumptions. Ideally, such constructions would avoid obfuscation altogether, but a more limited goal would be to get "correlated distributional indistinguishability" from indistinguishability obfuscation.

VI. Acknowledgments

The authors would like to thank Yael Tauman Kalai, Nir Bitansky and Omer Paneth for enlightening initial discussions on the topics of this work.

References

[AFK+14] Daniel Apon, Xiong Fan, Jonathan Katz, Feng-Hao Liu, Elaine Shi, and Hong-Sheng Zhou. Non-interactive cryptography in the ram model of computation. Cryptology ePrint Archive, Report 2014/154, 2014. http://eprint.iacr.org/.

[BCCT13] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. Recursive composition and bootstrapping for snarks and proof-carrying data. In Boneh et al. [BRF13], pages 111–120.

[BFR+13] Benjamin Braun, Ariel J. Feldman, Zuocheng Ren, Srinath T. V. Setty, Andrew J. Blumberg, and Michael Walfish. Verifying computations with state. *IACR Cryptology ePrint Archive*, 2013:356, 2013.

[BRF13] Dan Boneh, Tim Roughgarden, and Joan Feigenbaum, editors. *Symposium on Theory of Computing Conference, STOC'13, Palo Alto, CA, USA, June 1-4, 2013*. ACM, 2013.

[BSCG+13] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. Snarks for c: Verifying program executions succinctly and in zero knowledge. In Canetti and Garay [CG13], pages 90–108.

[BSCGT13] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, and Eran Tromer. Fast reductions from rams to delegatable succinct constraint satisfaction problems: extended abstract. In Robert D. Kleinberg, editor, *ITCS*, pages 401–414. ACM, 2013.

[CG13] Ran Canetti and Juan A. Garay, editors. *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part II*, volume 8043 of *Lecture Notes in Computer Science*. Springer, 2013.

[CIJ+13] Angelo De Caro, Vincenzo Iovino, Abhishek Jain, Adam O'Neill, Omer Paneth, and Giuseppe Persiano. On the achievability of simulation-based security for functional encryption. In *CRYPTO*, 2013.

[CKGS98] Benny Chor, Eyal Kushilevitz, Oded Goldreich, and Madhu Sudan. Private information retrieval. *J. ACM*, 45(6):965–981, 1998.

[CR73] Stephen A. Cook and Robert A. Reckhow. Time bounded random access machines. *J. Comput. Syst. Sci.*, 7(4):354–375, 1973.

[Gen09] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *STOC*, pages 169–178, 2009.

[GGH+13] Sanjam Garg, Craig Gentry, Shai Halevi, Mariana Raykova, Amit Sahai, and Brent Waters. Candidate indistinguishability obfuscation and functional encryption for all circuits. volume 2013, page 451, 2013. To appear in FOCS 2013.

[GHL+14] Craig Gentry, Shai Halevi, Steve Lu, Rafail Ostrovsky, Mariana Raykova, and Daniel Wichs. Garbled ram revisited. EUROCRYPT, 2014.

[GHRW14] Craig Gentry, Shai Halevi, Mariana Raykova, and Daniel Wichs. Outsourcing private ram computation. Cryptology ePrint Archive, Report 2014/148, 2014. http://eprint.iacr.org/.

[GKK+12] S. Dov Gordon, Jonathan Katz, Vladimir Kolesnikov, Fernando Krell, Tal Malkin, Mariana Raykova, and Yevgeniy Vahlis. Secure two-party computation in sublinear (amortized) time. In *CCS*, 2012.

[GKP+13a] Shafi Goldwasser, Yael Tauman Kalai, Raluca A. Popa, Vinod Vaikuntanathan, and Nickolai Zeldovich. How to run turing machines on encrypted data. In Canetti and Garay [CG13], pages 536–553.

[GKP+13b] Shafi Goldwasser, Yael Tauman Kalai, Raluca A. Popa, Vinod Vaikuntanathan, and Nickolai Zeldovich. Reusable garbled circuits and succinct functional encryption. In Boneh et al. [BRF13], pages 555–564.

[GO96] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, 1996.

[LO13a] Steve Lu and Rafail Ostrovsky. How to garble ram programs. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT*, volume 7881 of *Lecture Notes in Computer Science*, pages 719–734. Springer, 2013.

[LO13b] Steve Lu and Rafail Ostrovsky. How to garble RAM programs. Presentation in TCC 2013 rump session, 2013.

[OS97] Rafail Ostrovsky and Victor Shoup. Private information storage. In *STOC*, 1997.

[PF79] Nicholas Pippenger and Michael J. Fischer. Relations among complexity measures. *J. ACM*, 26(2):361–381, 1979.

[RAD78] Ron Rivest, Leonard Adleman, and Michael L. Dertouzos. On data banks and privacy homomorphisms. In *Foundations of Secure Computation*, pages 169–180, 1978.

[Val08] Paul Valiant. Incrementally verifiable computation or proofs of knowledge imply time/space efficiency. In Ran Canetti, editor, *TCC*, volume 4948 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2008.

[Yao82] Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In *FOCS*, pages 160–164, 1982.